A Flexible and Scalable Affinity Lock for the Kernel

Benlong Zhang, Junbin Kang, Tianyu Wo, Yuda Wang and Renyu Yang
State Key Laboratory of Software Development Environment
Beihang University
Beijing, P.R. of China
{zhangbl, kangjb, woty, wangyuda, yangry}@act.buaa.edu.cn

***Abstract*—A number of NUMA-aware synchronization algorithms have been proposed lately to stress the scalability inefficiencies of existing locks. However their presupposed local lock granularity, a physical processor, is often not the optimum configuration for various workloads. This paper further explores the design space by taking into consideration the physical affinity between the cores within a single processor, and presents FSL to support variable and finely tuned group size for different lock contexts and instances. The new design provides a uniform model for the discussion of affinity locks and can completely subsume the previous NUMA-aware designs because they have only discussed one special case of the model. The interfaces of the new scheme are kernel-compatible and thus largely facilitate kernel incorporation. The investigation with the lock shows that an affinity lock with optimal local lock granularity can outperform its NUMA-aware counterpart by 29.40% and 58.28% at 80 cores with different workloads.**

Keywords*—*synchronization algorithms, multi-core algorithms

I. Introduction

The overhead of synchronization mechanisms often drags system throughput behind in multiprocessing. The simple ticket spin lock, which is ubiquitously used in the kernel, is known to have innate scalability defects. There has been heated discussion on the design of scalable spin locks in the early 1990's, and a group of effective queue locks have been proposed (e.g., the MCS [1] lock). However, they often provide different APIs from the ticket lock for the kernel.

Recently a group of NUMA-aware lock schemes [9], [11], [12] have been proposed. Given the huge disparity of inter-chip cache access latency [3], [11], these two-level hierarchical designs are capable of improving synchronization scalability tremendously by the exploitation of socket-level lock data affinity. Though reasonable and effective, such locks simply presuppose the granularity of the first-layer local lock should be a single processor, which is often not the optimum configuration for different contexts. This approach have left out several important aspects for the design of affinity locks and stopped the exploration of the design space ahead of time.

As modern multi-processors exhibit continually increasing horizontal access latency for a certain core when the target private cache is becoming farther [3], the cost of synchronization also increases drastically with the number and span of cores involved. The access latency of the cache of a remote core residing on another processor can be as much as 100 times that of its own cache [3], thus the NUMA-aware locks divide the cores into several groups in the unit of a physical processor to lessen the inter-chip communication as much as possible. However, these constructs have neglected the still large access latency discrepancies among the cores within a single processor, because the latency to a nearby private cache can be as much as 40 times that of the local cache [3].

This paper extends the discussion of NUMA-aware locks and presents FSL: a flexible and scalable affinity lock for the kernel. The new lock design relies on the idea that the first-layer granularity of an affinity lock should be dynamically configured. It provides a uniform model to build affinity locks and explores the design space consistently and comprehensively. In this more general discussion, the group size in an affinity lock should be as small as possible to maximize the exploitation of the physical affinity of the cores as long as there are new waiters when the lock holder exits. For those locks with fierce competition intensity, a 2-core-based sharing can maximize performance; while for others, a 5-core-based sharing may be the best choice. The NUMA-aware locks discuss only one special case of the new scheme and can be completely subsumed into the new model.

Different from existing affinity locks providing uniform lock structure for every lock instance, FSL is capable of providing two instances with customized first-layer group size and thus heterogeneous construction according to their specific contexts. Considering the wide range of the number of processing cores within a physical processor, which can be as small as 2 in an Intel i3 and as big as 72 in a Tilera TILE-Gx72 processor, and the various contexts of the kernel locks, we believe our investigation is worthwhile. The new scheme requires constant memory space and exports compatible APIs with the ticket lock. In particular, the contributions of this work include: (1) the design and implementation of FSL, an affinity lock for the kernel with flexible first-layer granularity; (2) the investigation with FSL into different lock contexts and the conclusion that different lock instances should use different local lock granularity to maximize scalability.

II. FSL Design

Although existing NUMA-aware locks uniformly adopt a physical processor as the first-layer local lock granularity, the optimal group size of an affinity lock is yet still to be discussed. Considering the big on-chip core number of modern multicore processors and the tremendous access latency of private caches [3], contention of a local ticket lock may still considerably hinder lock performance.

The fundamental rationale of hierarchical affinity locks lies in the global lock will be hold for a while until relinquished whereas the local lock is frequently acquired and released within the group. The global lock should be released under two circumstances: it has been hold for enough time or there

```
1  #define CORE_COUNT  80
2  #define CPU_SOCKETS 8
3  #define MAX_GROUPS  (CORE_COUNT / 2)
4  #define CACHE_LINE_SIZE   128
5  #define OVERFLOW  (1<<(sizeof(__ticket_t)*8))
6  #define CORE_THRESHOLD    25
7
8  typedef struct local_ticket {
9    struct __raw_ticket {
10     __ticket_t head, carry, pad, tail;
11   } ticket;
12   u8 pad[CACHE_LINE_SIZE - 4*sizeof(__ticket_t)];
13 } local_ticket_t
14 __attribute__((__aligned__(CACHE_LINE_SIZE)));
15
16 typedef struct fsl_spinlock {
17   int                     group_size;
18   __attribute((__aligned__(CACHE_LINE_SIZE)));
19   int                     core
20   __attribute((__aligned__(CACHE_LINE_SIZE)));
21   /* global lock; */
22   struct __raw_ticket     gl_ticket
23   __attribute((__aligned__(CACHE_LINE_SIZE)));
24   /* local locks; */
25   local_ticket_t          bl_tickets[MAX_GROUPS];
26 } fsl_spinlock_t;
27
28 #define FSL_SPINLOCK_UNLOCKED(count) \
29         {.group_size=count}
```

Fig. 1: **Lock Definition.** *The figure above gives the structure of the new lock on an 80-core machine with 8 sockets.*

are no more waiters in this group. It is intuitive that the smaller a group is, the better the locality will be as the cores are physically closer to each other. But, contradictorily, the group size must be big enough to guarantee the group will have accumulated other waiters at the exit of the current holder to avoid the frequent release of the global lock which will destroy the foundation of FSL and thus drag performance down.

A. Flexible Affinity

We believe the group size of FSL should be decided flexibly by the arrival rate of new comers in this group. As different lock instances have different new waiter arrival rate and contention intensity, FSL should flexibly provide customized group size for them. As Figure 1 displays, the type `fsl_spinlock_t` defines the structure of the new lock. Its first cache line contains the group count of the lock and is specified at initialization. The second line contains the identifier of the latest core that has just successfully acquired this lock instance. The third line stores the global ticket and the rest are ticket locks of the individual local locks. Note the group count should be between `CPU_SOCKETS` and `MAX_GROUPS`.

Each local lock is a ticket lock that occupies a single cache line. Apart from the `head` and `tail` fields (usually two bytes long each) with the same usage as in the ticket lock, the `carry` variable is inserted to protect `tail` from being contaminated by the carry of `head`, the process of which will be detailed later; and the extra `pad` is filled in for address alignment.

B. Two-Layered Locking

FSL adopts a two-layered locking scheme. The first layer, privately owned on a group basis, is designed to transfer possession of the lock between the cores of the group. And the second layer, which is also a ticket lock, is devised to pass ownership of the lock among the groups globally. The `gl_ticket` is the second layer ticket lock.

```
1  static __always_inline
2  void fsl_lock (fsl_spinlock_t *lock)
3  {
4    int core, group, threshold;
5    socket_ticket_t *line;
6    struct __raw_ticket inc = {.tail=1};
7    struct __raw_ticket incx = {.tail=1};
8
9    //(1) locate the group;
10   core = task_cpu (current);
11   threshold = CORE_THRESHOLD * lock -> group_size;
12   group = core / lock -> group_size;
13   line = &lock -> bl_tickets[group];
14
15   //(2) acquire the local lock;
16   inc = xadd(&line->ticket, inc);
17
18   //(3.1) acquire and wait for the global lock;
19   if (inc.head == inc.tail ||
20     inc.tail % threshold == 0) {
21     incx = xadd (&lock -> gl_ticket, incx);
22     while (1) {
23       if (incx.head == incx.tail) break;
24       cpu_relax ();
25       incx.head = ACCESS_ONCE
26           (lock -> gl_ticket.head);
27     }
28   }
29
30   //(3.2) wait for the local lock;
31   else {
32     while(1) {
33       if (inc.head == inc.tail) break;
34       cpu_relax();
35       inc.head = ACCESS_ONCE(line -> ticket.head);
36     }
37   }
38
39   //(4) record the core number;
40   lock -> core = core;
41 }
```

Fig. 2: **Lock Acquisition.** *The xadd instruction is the fetch_and_inc.*

Figure 2 details the procedures to go through to acquire the lock. First, the thread finds out the group it's running on according to the current core identifier. For the platforms where consecutive core numbers are physically consecutive, as shown in the figure, the thread gets its group by dividing the core number with `group_size`. For those platforms where interleaved core numbers are physically close (e.g., core 0, 4, 8 and 12 may belong to a processor in a 16-core machine), this step will be different. Second, the thread takes a seat in the local ticket by executing the `fetch_and_inc` instruction `xadd`. Third, if the returned value indicates the thread is the first comer, it will take another seat in the global ticket and wait in a tight loop until it's granted. Non-first comers will enqueue themselves to wait for the local ticket lock. At last, the new lock holder will update `core` when it exits the loop.

Similarly, as shown in Figure 3, to release a lock the lock holder should first find out the target group with `core`. Notice we cannot get the core identifier at this time with `task_cpu` as the thread might have already been migrated to a different core. Then the local ticket ownership is relinquished by increasing the `head` field. The returned snapshot of the local ticket tells the state of the local lock. And if there are more waiters on it `tail` should be at least bigger than `head` by two, otherwise the `global_release` flag will be set thus the inter-group lock can be released. Note that the carry of `head` should be zeroed once it overflows (in line 20).

As stated above, the effectiveness of the design is largely based on the assumption that there will be waiters on the socket

```
static __always_inline
void fsl_unlock(fsl_spinlock_t *lock)
{
  int core, group_count, head_rounded;
  int threshold, global_release = 0;
  socket_ticket_t *line;
  struct __raw_ticket inc = {.head = 1};

  //(1) locate the group;
  core = lock -> core;
  threshold = CORE_THRESHOLD * lock -> group_size;
  group_count = core / lock -> group_size;
  line = &lock -> bl_tickets[group_count];

  //(2) release the local lock;
  inc = xadd (&line -> ticket, inc);
  head_rounded = (inc.head + 1) % OVERFLOW_BOUND;

  /* clear the carry; */
  if (!head_rounded)
    line -> ticket.carry = 0;
  /* no more waiters in this group; */
  if (head_rounded == inc.tail)
    global_release = 1;
  /* fairness threshold; */
  if (head_rounded % threshold == 0)
    global_release = 1;

  //(3) release the global lock;
  if (global_release) {
    __add (&lock -> gl_ticket.head, 1,
      UNLOCK_LOCK_PREFIX);
  }
}
```

Fig. 3: **Lock Release.** *The __add instruction increases the head of the inter-socket ticket by 1.*

ticket when the holder tries to release the lock, thus the global ticket lock can be hold for a while by the processor before it is relinquished. The code segment from line 30 to 33 in Figure 3 is expected to be rarely executed.

C. Fairness

Evidently the global lock cannot be permanently hold until this group becomes empty as other groups may be left starving. Straightforwardly, relinquishment on threshold can be an effective strategy for fairness assurance: each group keeps accounting the times that the lock has been acquired and then voluntarily give up once the threshold is reached. As indicated by line 19 and 20 in Figure 2, the new comer at the threshold value automatically forces itself to wait for the global ticket lock. Correspondingly, line 26 in Figure 3 shows that a thread will judiciously choose to release the global lock on the finding that the granted times of this group has arrived at the threshold, which is indicated by the current `head` value of the local lock.

III. Evaluation

Two benchmarks with disparate critical section length and consecutive acquisition interval are selected to illustrate the flexibility of our new lock. All the experiments have been carried out with linux-kernel-3.12.2 on an Supermicro server, as detailed in Table I.

TABLE I: Experimental Platform.

Vendor	Supermicro	L1 D cache	32KB
Processors	8×Intel Xeon E7-8870	L1 I cache	32KB
# of Cores	80	L2 cache	256KB
Frequency	2.40GHz	LLC	30MB

Scalability: A standard measurement for the scalability of a lock design is to measure the time of `N` threads collectively executing `C` lock acquisitions and releases, with each thread allocated `C/N` times. Then we can get the average time for a single acquisition and release of the lock under different contention degree, which is characterized by the number of threads. A lock construct with satisfactory scalability should keep the time constant when varying the number of threads.

The system call `sys_spinlock(mode, count)` is added to initiate `count` lock acquisitions and releases of different locks. The total count `C` is configured as 1 million, and we bind each thread to a core to generate reproducible results. There are three observations on the results in Figure 4. First, of all the FSL configurations, FSL-1 and FSL-80 exhibit the worst performance similar to that of the ticket lock. From 80 to 2, the lock performance improves with the group size becoming smaller. FSL-2, FSL-5 and FSL-10 outperform MCS. Second, the 2-core-based FSL achieves the best performance as it's intensive contention leads to the short interval. It outperforms the 10-core-based FSL and MCS by 29.40% and 55.55% at 80 cores. Third, the performance of the ticket lock is closely related to hardware affinity, which is characterized by the fact that consecutively adding threads to a processor does not increase the time but adding processors will, and resulted in the gaps in its result line. The increased latency of all FSL configurations are similar, being about 104% of the ticket lock due to the two-layered locking.

Throughput: We adopt the benchmark `dup-close` in the motivation of the corey [3] paper as a survey of the potential of FSL on applications. Each thread of the multi-threaded benchmark creates a file descriptor, then repeatedly duplicates it and closes the result. The threads are also bound to cores, and the throughput is measured. In this configuration both the critical section length and the interval are significantly larger.

Figure 5 presents the results. Of all the FSL locks, the 5-core-based FSL yields the maximum throughput, which outperforms FSL-10 by 58.25% at 80 cores. It can be seen that performance of FSL-5 is not as good as the ticket lock within 5 cores, but once the threads span to two groups, ticket lock quickly deteriorates and its throughput remains unacceptable afterwords. The throughput of FSL-5 remains horizontal and suffers little decline with the number of cores increasing, and outperforms the ticket lock by 14.65x at 80 cores.

IV. Related Work

Centralized Locks: The most intuitive spin lock is simply comprised of a flag variable [1], [2], with each core repeatedly executing the `test_and_set` instruction trying to acquire the lock. To cut off the blind competition, each waiter can first check the lock state and then carry out the `test_and_set` if it's released, which is known as the `test-and-test_and_set` [1], [2] technique. Furthermore, delays, whether being static or dynamic, can be inserted into various locations of the lock acquisition algorithm to reduce collision [2]. The ticket lock [10] is centralized as the two counters incorporated are typically fit into a single cache line. As have been extensively investigated, these locks have poor scalability due to their centralized designs.

Queue Locks: The motivation and fundamental rationale of the queue locks is to distribute the lock data so that each waiter can have its own locally accessible data. The

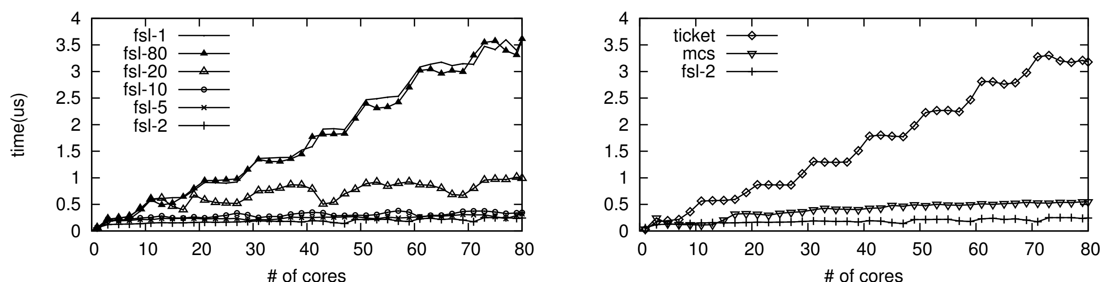

Fig. 4: **Scalability Test Result.** *The even core points are omitted (except 80) for display clearness and clarity.*

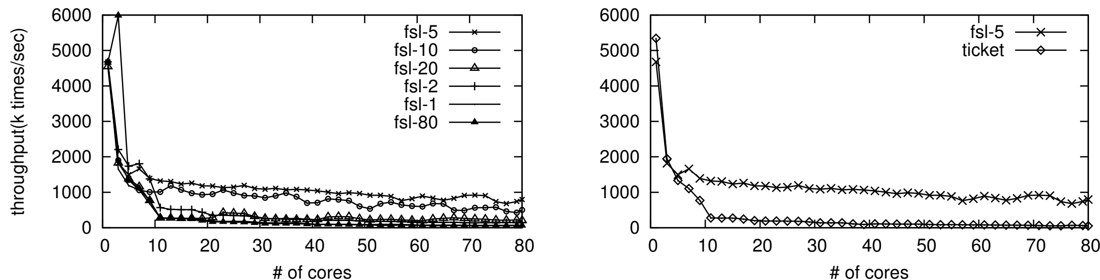

Fig. 5: **Throughput of `dup-close`.** *The even core points are omitted (except 80) for display clearness and clarity.*

Anderson [2] lock, and the Graunke and Thakkar's [5] lock are both composed of a fixed number of slots, with each slot prepared for a single thread by sequencing techniques. A major defect of them, however, is the tremendous space requirement proportional to the number of threads. Moreover, the fixed slot size largely casts doubt on their practicality. The MCS lock [1] represents each waiter with a `qnode` structure from the stack of the invoker and delivers satisfactory scalability. K42 [12] is a variant to make up the interface incompatibility of MCS, but incurs overhead by a significant margin due to its increased complexity [4]. The CLH lock [7] is a variant of MCS where the waiter spins on its predecessor `qnode`.

NUMA-aware Locks: The cache coherent NUMA nature of modern multiprocessing machines suggests lock designs with locality in consideration may have the chance to further boost performance. HCLH [8] is a hierarchical variant of CLH that collects requests on each chip into a local queue and then has them integrated into the global queue. Dave et al. propose the FC-MCS [9] hierarchical algorithm based on the flat combining technique and MCS. Then the lock cohorting model [12] is subsequently proposed for general transformation from existing NUMA-oblivious locks to NUMA-aware locks. Tudor et al. [11] observe that performing any operation on a cache line crossing sockets does not scale and investigate the potential of hierarchical ticket lock.

V. Conclusions

This paper proposes that the granularity of an affinity lock should be flexibly decided according to the contexts of the lock instances. Then FSL is designed and implemented for the Linux kernel. The investigation concludes that different lock instances should adopt different granularities to maximize scalability. This work provides a generalized model for the discussion of affinity locks.

VI. Acknowledgments

The work was funded by China 973 Program (No.2011CB 302602), China 863 Program (No. 2013AA01A213), HGJ Program (2010ZX01045-001-002-4, 2013ZX01039-002-001-001), Projects from NSFC (No.61170294, 91118008) and Fundamental Research Funds for the Central Universities. Tianyu Wo is the corresponding author.

References

[1] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems, 9(1):21-65, 1991.

[2] Thomas E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Transactions on Parallel and Distributed Systems, 1(1):6-16, Jan. 1990.

[3] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, Zheng Zhang. Corey: an operating system for many cores. In Proceedings of 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08), December, USA.

[4] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris and N. Zeldovich. Non-scalable locks are dangerous. MIT CSAIL.

[5] Gary Graunke and Shreekant Thakkar. Synchronization algorithms for shared-memory multiprocessors. Computer 23(6):60-69.

[6] Peter, Anders and Erik. Queue locks on cache coherent multiprocessors. Parallel Processing Symposium, 1994.

[7] T. S. Craig. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report UW-CSE-93-02-02, University of Washington.

[8] V. Luchangco, D. Nussbaum and N. Shavit. A hierarchical CLH queue lock. Proceedings of the European Conference on Parallel Computing, August-September 2006. Dresden, Germany.

[9] D. Dice, V. Marathe, and N. Shavit. Flat Combining NUMA Locks. In Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures, 2011.

[10] kernel ticket spin lock. Linux source:arch/x86/include/asm/spinlock.h.

[11] Tudor David, Rachid Guerraoui, Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In Proceedings of the 24th ACM Symposium on Operating Systems Principles. November 3-6, 2013, Nemacolin Woodlands Resort, Farmington, Pennsylvania, USA.

[12] D. Dice, V. Marathe, and N. Shavit. Lock cohorting: a general technique for designing numa locks. 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming(PPoPP), pages 247-256, 2012.